Lecture F:

LL & Recursive Descent Parsing

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- HW2 due tomorrow night
- Parser/AST Project
  - due next Thursday
  - Important to show up to section tomorrow!
- Mini HW 3 out tomorrow or Friday (only one late day allowed)
- More on LL grammars and HW3 next week’s section

Outline

- Top-down Parsing
- LL(k) Grammars
- Recursive Descent
- Hacking Grammars to Work Top-Down
  - Left Recursion
  - Common Prefixes
- What Do Real Compilers/Interpreters Do?

Outline

Top-down Parsing

LL(k) Grammars

Recursive Descent

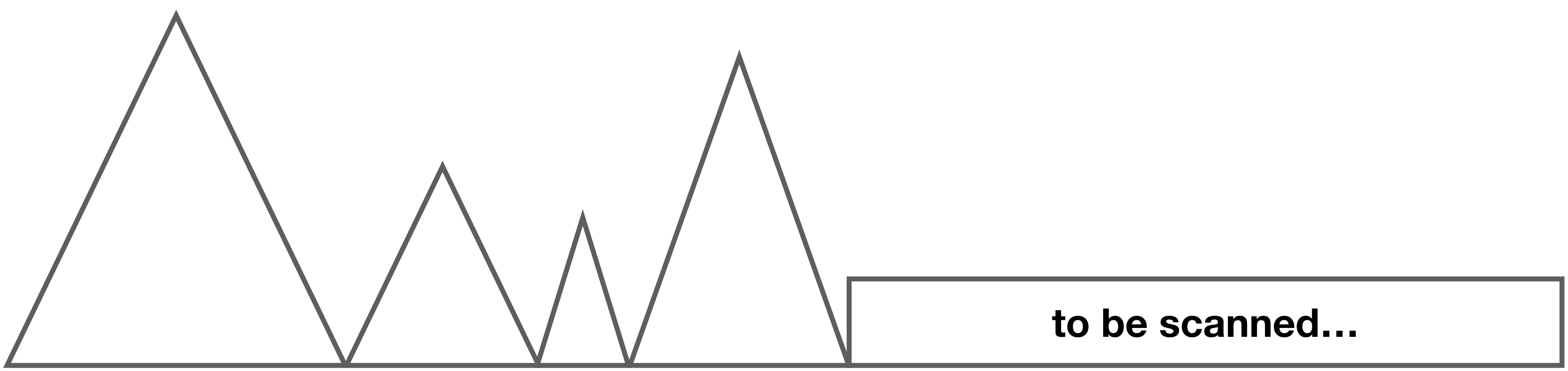
Hacking Grammars to Work Top-Down

- Left Recursion
- Common Prefixes

What Do Real Compilers/Interpreters Do?

The Bottom-Up Approach

- Build up the tree from the leaves
  - ✦ Shift next input or reduce using a production
  - ✦ Accept when all input has been read and reduced to the start symbol of the grammar
- LR(k) and subsets thereof (SLR, LALR(k), …)

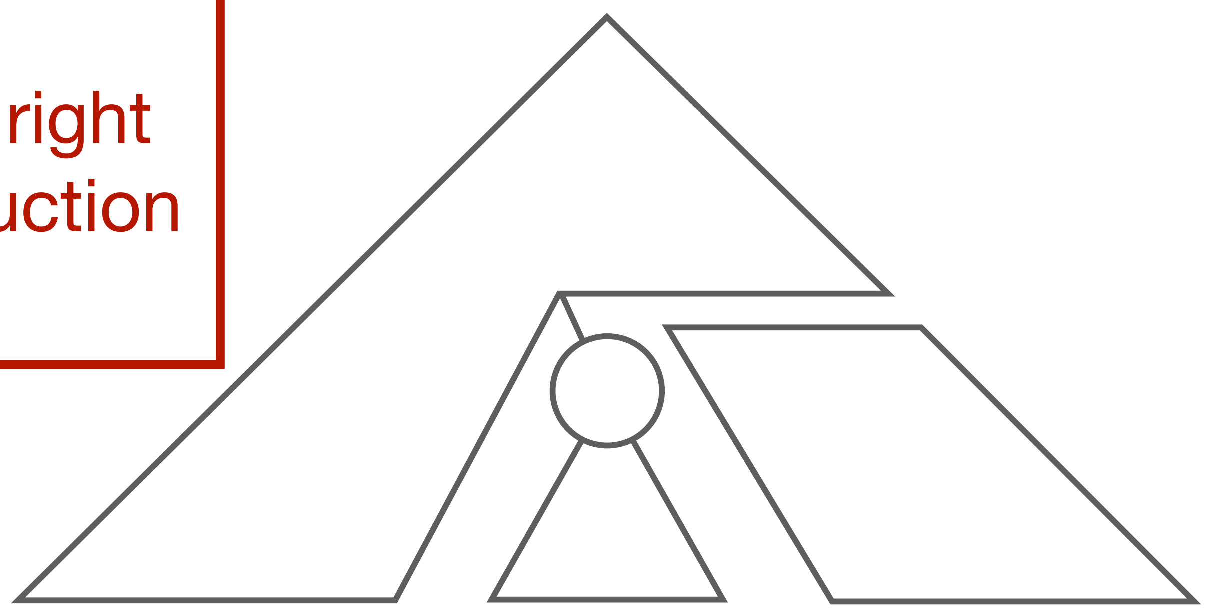


The Top-Down Approach

- Begin at the root with the start symbol of the grammar
  - ✦ Repeatedly pick a non-terminal and expand
  - ✦ Accept when expanded tree matches the input
- LL(k)

Problem

How do we know the right choice of which production to expand with?



Left-most Derivations

- The top-down parse will be a **left-most derivation**

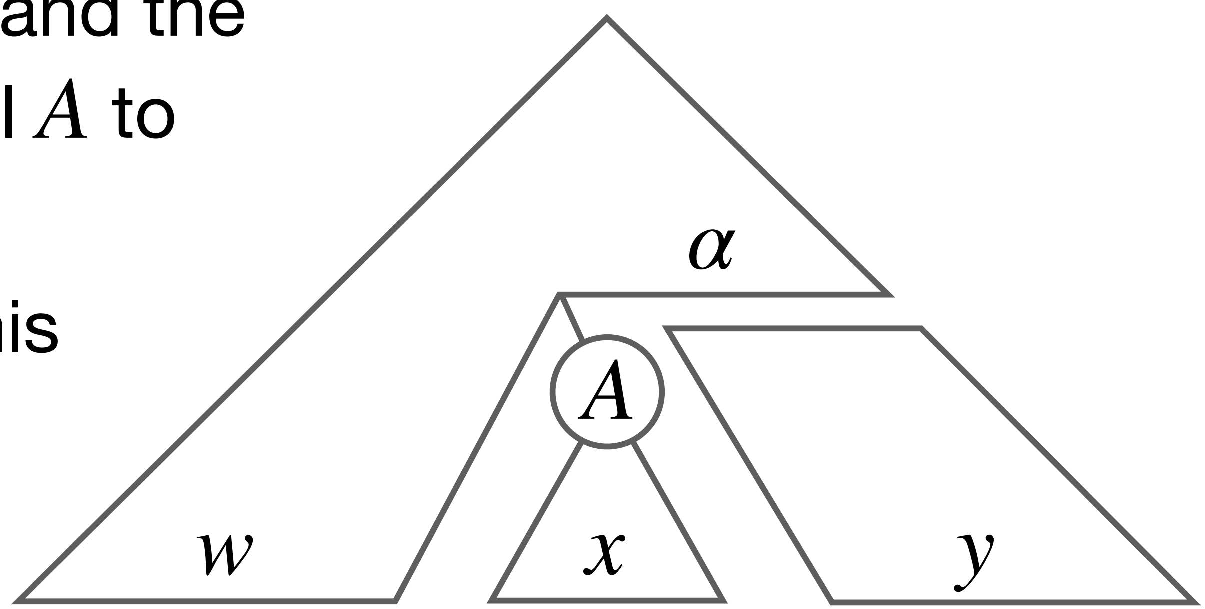
$$S \Rightarrow_{\text{lm}} wA\alpha \Rightarrow^{*}_{\text{lm}} wxy$$

- At each step, pick some production

$$A ::= \beta_1\beta_2\cdots\beta_n$$

  that will properly expand the leftmost non-terminal A to match the input

- How can we make this choice deterministic (i.e. no backtracking)



Predictive Parsing

- If we are expanding at some non-terminal A, and there are two or more possible productions for A

  $A ::= \alpha$

  $A ::= \beta$

  then we want to make the correct choice by looking at just ***the next*** input symbol
- If we can do this, we can build a **predictive parser** that can perform a top-down parse without backtracking

Example — How can we predict?

- Seems impossible, but programming language grammars are often suitable for predictive parsing (by design!)
- Typical example

$$
\begin{aligned}
stmt ::= &\ id \texttt{ = } exp \texttt{ ;} \\
| &\ \texttt{return } exp \texttt{ ;} \\
| &\ \texttt{if (} exp \texttt{) } stmt \\
| &\ \texttt{while (} exp \texttt{) } stmt
\end{aligned}
$$

- If the next part of the input begins with the tokens

  IF LPAREN ID(x) …

  then we should expand *stmt* to an if-statement

Outline

Top-down Parsing

LL(k) Grammars

Recursive Descent

Hacking Grammars to Work Top-Down

- Left Recursion
- Common Prefixes

What Do Real Compilers/Interpreters Do?

LL(1) Property

- Def. A grammar has the LL(1) property when, for all non-terminals A, and distinct* productions $A ::= \alpha$ and $A ::= \beta$, it is the case that
  - $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
    - *(intuitively, if the lookahead is x and $x \in FIRST(\alpha)$, then derive α. If the lookahead is x and $x \in FIRST(\beta)$, then derive β.)*
  - $NULLABLE(A) \implies FIRST(\alpha) \cap FOLLOW(A) = \emptyset$
    - *(If the lookahead is x, A is nullable, and $x \in FOLLOW(A)$, then derive ϵ. Otherwise if $x \in FIRST(\alpha)$, then derive α.)*
- If a grammar has the LL(1) property, then we can build a predictive parser for it that uses 1 symbol of lookahead

***distinct here means** $\alpha \neq \beta$

LL(k) Parsers

- An LL(k) parser
  - read the input — **Left-to-right** not right-to-left
  - derivation order — will produce a **Leftmost** derivation
  - Looking ahead at most **k** terminal symbols
- 1-symbol lookahead is enough for many practical programming language grammars
  - LL(k) for k > 1 is rare in practice…
  - and violations of 1 lookahead are sufficiently rare that you can just “cheat” with more lookahead where needed in a hand-written parser

Table-Driven LL(k) Parsers

- As with LR(k), a table-driven parser can be constructed from the grammar
- A very simple LL(1) example…

    1. $S ::= (\ S \ ) \ S$
    2. $S ::= [\ S \ ] \ S$
    3. $S ::= \epsilon$

- Table (one row per non-terminal showing which production to apply given the next input symbol)

	(	)	[	]	$
S	1	3	2	3	3

LL vs. LR

- LR is more powerful than LL (formally)
  - LL has to make a decision based on the current non-terminal and lookahead alone
  - LR can make a decision based on the entire stack contents as well as lookahead
- Tools can generate parsers for LL(1) and for LR(1) grammars
  - (editorial) so you might as well use an LR parser gen.
  - *Caveat* — a parser generator tool with a better community, documentation, support, and error messages might be a better choice even if LL-based

Outline

Top-down Parsing

LL(k) Grammars

Recursive Descent

Hacking Grammars to Work Top-Down

- Left Recursion
- Common Prefixes

What Do Real Compilers/Interpreters Do?

Recursive Descent Parsers

- Top-down parsing is easy to implement by hand
  - ✦ Earliest parser type still in major use (CACM Jan. 1961)
  - ✦ Implementations are much more human-readable than generated, table-driven parsers
- Key Idea — write one procedure (function, method) corresponding to each major non-terminal in the grammar
  - ✦ Each of these methods is responsible for matching its non-terminal with the next part of the input
  - ✦ Like structural recursion, but patterned on the *output*, (really, on the grammar) rather than the input to the parsing pass

Example – Statements

```
StmtNode parseStmt() {
  switch(nextToken) {
    ID: var id = parseId();
        match(EQ);
        var exp = parseExp();
        match(SEMICOLON);
        return new AssignNode(id, exp);

    IF: match(IF); match(LPAREN);
        var exp = parseExp();
        match(RPAREN);
        var stmt = parseStmt();
        return new IfNode(exp, stmt);

    WHILE: …
    RETURN: …
  }
}
```

$$
\begin{aligned}
stmt ::= \; & id \; \texttt{=} \; exp \; \texttt{;} \\
\mid \; & \texttt{return} \; exp \; \texttt{;} \\
\mid \; & \texttt{if} \; (\; exp \;) \; stmt \\
\mid \; & \texttt{while} \; (\; exp \;) \; stmt
\end{aligned}
$$

From Theory to Practice…

- Observe — the pattern of method calls here reflects the leftmost derivation in the parse tree
- The example on the last slide has some deficiencies
  - Error reporting — How should errors be handled?
  - (tricky to get right) — how can/should you recover from parse errors, so that you can continue a best-effort parse?

Invariant for Parser Functions

- The different functions within the parser need to agree on a convention for where the scanner token stream should be before and after calling a function
- A good choice of invariant — When a parser function is called, the current token (next unprocessed piece of the input) is the token that begins the expanded non-terminal being parsed
  - Corollary — when a parser function is done, it must have completely consumed the input corresponding to the non-terminal it is responsible for parsing

Outline

Top-down Parsing

LL(k) Grammars

Recursive Descent

Hacking Grammars to Work Top-Down

- Left Recursion
- Common Prefixes

What Do Real Compilers/Interpreters Do?

2 Problems for Top-Down Parsers

- **Left Recursion** in the grammar
  - e.g. $expr ::= expr + term \mid term$
  - note: left recursion is very important for expressing left-associative operators (most binary operators) — so this is a big problem we need to solve
- **Shared prefixes** among different productions
  - e.g. Stmt ::= id = exp ; | id += exp ;
  - note: this grammar is not ambiguous or complicated to parse. We just have to defer till after id to disambiguate

Outline

- Top-down Parsing
- LL(k) Grammars
- Recursive Descent
- Hacking Grammars to Work Top-Down
    - **Left Recursion**
    - Common Prefixes
- What Do Real Compilers/Interpreters Do?

The Left Recursion Problem

```
ExprNode parseExpr() {
  var expr = parseExpr();
  match(PLUS);
  var term = parseTerm();
  return AddNode(expr, term);
}
```

$$
\begin{aligned}
expr ::= &\ expr + term \\
\mid &\ term
\end{aligned}
$$

Great code, right?

A Solution to Our Problem?

- Use right recursion instead!

$$expr ::= term + expr \mid term$$

- Will this work right?

- Problem — we will not get left-associativity any more
    - (sometimes the associativity doesn't matter, but if it does…)

A Formal Solution

- Rewrite using right recursion and a new non-terminal
- Original grammar

$$expr ::= expr + term \mid term$$

- New grammar

$$expr ::= term \ exprtail$$

$$exprtail ::= + \ term \ exprtail \mid \epsilon$$

- Properties
  - ✦ No infinite recursion when coded directly
  - ✦ Not entirely obvious how this produces left-associativity

Another View on This Solution

- Observe that our original grammar

  $$expr ::= expr + term \mid term$$

  only generates finite sequences of the form

  $$(\cdots((term + term) + term) + \cdots) + term$$

- So, if we allow for using the Kleene star as sugar in our grammar, then we can instead express the same fix as

  $$expr ::= term\ \{+\ term\}^*$$

- This expression more directly leads to code for use in our recursive-descent parser

Fixed Recursive Descent Code

```
ExprNode parseExpr() {
  var term = parseTerm();
  var expr = term;
  while (nextToken == PLUS) {
    match(PLUS);
    var term = parseTerm();
    expr = AddNode(expr, term);
  }
  return expr;
}
```

$$expr ::= term\ \{+\ term\}^*$$

Indirect Left Recursion

- There are more insidious forms of left-recursion, e.g.

$$A ::= Bc$$
$$B ::= Ad \mid \epsilon$$

- Solution — (step 1) transform the grammar to one where all productions are either

$A ::= x\alpha$ (starts with a terminal symbol)

$A ::= A\alpha$ (rule has direct left recursion)

then (step 2) use our preceding trick to eliminate all direct left recursions from the grammar

Eliminating Indirect Left Recursion

- Basic idea — rewrite all productions $A ::= B\beta$ where A and B are different non-terminals by using all $B ::= \ldots$ productions to create new productions replacing the B in the $A ::= B\beta$ production — i.e. we **inline** the B productions
  - If there is an indirect cycle, this converts it to a direct cycle
- e.g. original

$$A ::= Bc$$
$$B ::= Ad \mid \epsilon$$

- converted

$$A ::= Adc \mid c$$
$$B ::= Ad \mid \epsilon$$

Outline

- Top-down Parsing
- LL(k) Grammars
- Recursive Descent
- Hacking Grammars to Work Top-Down
    - Left Recursion
    - **Common Prefixes**
- What Do Real Compilers/Interpreters Do?

Common Prefixes — Left Factoring

- If two rules for a non-terminal A have right hand sides that begin with the same symbol, then we can't predict which one to use. e.g.

$$stmt ::= \ id = expr \ ; \ | \ id \mathrel{+}= expr \ ;$$

- Formal solution — factor out the common prefix into a separate production. e.g.

$$stmt ::= \ id \ assign$$
$$assign ::= \ = expr \ ; \ | \ \mathrel{+}= expr \ ;$$

  - ✦ The non-terminal *assign* can now distinguish the two cases by inspecting the first token

Example — Parser Code

$$\begin{aligned} \textit{stmt} &::= \textit{id assign} \\ \textit{assign} &::= \; = \textit{expr} \; ; \;\mid\; += \textit{expr} \; ; \end{aligned}$$

```
StmtNode parseStmt() {
  var id = parseId();

  boolean reduce = false;
  if (nextToken == EQ) {
    match(EQ); reduce = false;
  } else if (nextToken == PLUSEQ) {
    match(PLUSEQ); reduce = true;
  }

  var exp = parseExp();
  match(SEMICOLON);
  if (reduce)
    return new ReduceNode(id, exp);
  else
    return new AssignNode(id, exp);
}
```

Outline

- Top-down Parsing
- LL(k) Grammars
- Recursive Descent
- Hacking Grammars to Work Top-Down
  - Left Recursion
  - Common Prefixes
- **What Do Real Compilers/Interpreters Do?**

Real Parsers for Major Languages

- Glossary of terms
  - **Handwritten** — some variant of recursive descent, usually with some idiosyncrasies / “cheating”
  - **YACC-like Parser Generator** — YACC, Bison, ANTLR, CUP, etc.
  - **PEG (Parsing Expression Grammars) or Parser Combinators** — a formalism for expressing only unambiguous grammars; a very different kind of parser generator than the ones we studied

Data on (Some) Major Languages

Handwritten

- C (GCC, Clang)
- Javascript (V8)
- Typescript
- CSS (Chromium)
- Java (OpenJDK)
- .NET (Roslyn)
- Golang
- Lua
- Swift
- Julia

PEG

- Python (CPython)

Yacc-like Parser Generator

- Ruby
- PHP (Zend Engine)
- Bash
- R
- SQL (Postgres, MySQL, SQLite)

source: Phil Eaton https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html

Practical Considerations

- IDEs (Integrated Development Environments) and the **Language Server Protocol**
  - In order to build tools that interactively analyze source code in IDEs, it's often necessary to ***parse*** that source code
- Problem — code in the middle of being edited is probably not grammatical
  - Thus, good parsers should be ***interactive*** and ***tolerant to errors***. Parser error recovery is essential
- Good parser error messages make a big difference!

Onwards! and Downwards!

- We're done with parsing!
- Rest of this week and next
  - Checking — make sure the program is valid
  - Symbol tables — the two hardest problems in CS are?
  - IRs — how should we represent code